Microprocessors and Microsystems 37 (2013) 472–481

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Hardware support for CSP on a Java chip multiprocessor

Flavius Gruian [a,*], Martin Schoeberl [b]

[a] Dept. of Computer Science, Lund University, 22100 Lund, Sweden
[b] Dept. of Informatics and Mathematical Modeling, Technical University of Denmark, 2800 Lyngby, Denmark

ARTICLE INFO

Article history:
Available online 30 August 2012

Keywords:
Java
Embedded systems
Network-on-chip
Communicating sequential processes

ABSTRACT

Due to memory bandwidth limitations, chip multiprocessors (CMPs) adopting the convenient shared memory model for their main memory architecture scale poorly. On-chip core-to-core communication is a solution to this problem, that can lead to further performance increase for a number of multithreaded applications. Programmatically, the Communicating Sequential Processes (CSPs) paradigm provides a sound computational model for such an architecture with message based communication. In this paper we explore hardware support for CSP in the context of an embedded Java CMP. The hardware support for CSP are on-chip communication channels, implemented by a ring-based network-on-chip (NoC), to reduce the memory bandwidth pressure on the shared memory.

The presented solution is scalable and also specific for our limited resources and real-time predictability requirements. CMP architectures of three to eight processors were implemented and tested on both Altera (EP1C12, EP2C70) and Xilinx (XC3S1200e) FPGAs, showing that the NoC accounts for under 9% of the total device area used by the system. Compared to shared memory-based communication, our NoC-based solution is between 1.7 and 9.3 times faster for raw data transfer, depending on the communication and memory configuration. Application speed-up, on the other hand, is highly dependent on the type of processing, as our measurements show.

1. Introduction

Chip multiprocessors (CMPs) are gradually replacing single core processors in all kinds of systems, as technology can no longer keep pace with the requirements of today's applications in terms of performance and power consumption. Chip manufacturers are hopeful that more cores automatically means higher performance, as more threads can execute in parallel. Traditionally, the programming model for threads executing on single core is usually based on shared memory, which imposes relatively few problems on such architectures. Nevertheless, the shared memory model runs into problems as the number of cores increases, since memory bandwidth and cache coherence issues become limiting factors. Although considerable effort is spent on providing a cache-coherent view of shared memory, also with non-uniform memory access time (ccNUMA), we consider that (additional) message passing is the path to further multi-core performance increase. To overcome the shared memory issues and provide further performance increase as systems scale up, two important changes must be adopted: better core-to-core communication mechanisms, e.g., networks-on-chip (NoC), and suitable programming models, based on message passing, e.g., Communicating Sequential Processes (CSPs).

Attempts to exploit thread-level parallelism through both architecture and programming model are not new, and had been tentatively investigated in the past, such as the Transputer/Occam approach [1,2]. Nevertheless, at the time when Transputers were introduced, single core performance started to increase about 52% per year [3]. Conventional CPU designs were offering enough performance and therefore little incentive to rewrite code to expose parallelism, rendering the Transputer approach unsuccessful at the time.

Recently, as single cores cannot offer enough performance, Transputer-like architectures are again coming into focus in the form of massively parallel processing arrays (MPPAs). These are offering both a high degree of parallelism and a scalable communication structure [4–6]. Naturally, the computational models used for programming such architectures are Kahn process networks (KPNs [7]) or communicating sequential processes (CSPs [8]).

In this paper we present an approach for relaxing the memory bandwidth pressure in a embedded Java CMP [9], by means of hardware communication channels and a CSP programming model for Java similar to [10–12]. The initial system, containing a shared memory, is augmented with a network-on-chip (NoC), intended to provide a direct way of exchanging data between processors. Shared memory is still used to store the code, as many of the Java

* Corresponding author.
E-mail addresses: flavius.gruian@cs.lth.se (F. Gruian), masca@imm.dtu.dk (M. Schoeberl).

0141-9331/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2012.08.004

classes must usually be available to the majority of tasks running on different cores. Furthermore, shared memory may be used and is the default mechanism for storing data and exchanging data between processors. The new addition is a ring network-on-chip, designed for predictability and minimal device utilization. The network is accessible from each core via custom routers that have an easy-to-use interface. Scratch-pad memories are employed to store network-passed data, in order to avoid the need to use shared memory, which would negate our initial intention. A specific application programming interface (API), in the form of Java classes, has been written to allow CSP-like programming to take advantage of underlaying architecture.

This paper is an extension of [13]. The current version contains a number of additions, improvements and updates. It introduces a novel multi-ring network architecture, where several rings may be connected through switches, in order to accommodate large networks, without the risk of excessive delays between selected nodes. We also describe experiments carried out on larger systems, of up to eight cores, on yet another FPGA platform (Altera DE2). Additionally, we give performance measurements for some more realistic applications, in addition to the simple inter-processor data transfer via shared memory and NoC. Finally, we rerun the experiments to use the latest version of hardware, core updates, and tools. The contributions of this paper are as follows:

1. Hardware support for CSP channels with a network-on-chip
2. A time-predictable network-on-chip
3. Java based API for CSP communications
4. Multi-ring architecture for larger systems

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 describes the driving requirements for our system. The hardware solution is described in detail in Section 4 while the software architecture is addressed in Section 5. Evaluation results of our design are presented in Section 6 and conclusions are drawn in Section 7.

2. Related work

The Communicating sequential processes (CSPs [8]) model was first introduced by C.A.R. Hoare in 1978 and has since seen a few changes and many successful applications, influencing a number of related models of computation. At the basis of CSP are processes (fundamental behaviors) operating independently and interacting with each other only through events (messages). The ways these processes may be composed (sequential, parallel, alternative, etc.) and how they communicate with each other or the environment are precisely defined through algebraic operators. This formal basis made CSP appealing to a wide range of domains, including specification, modeling, verification and analysis of various complex systems (hardware, dependable and safety–critical systems, protocols, etc.).

Occam [14] as a programing and specification language and the Transputer [1,2] designed to run Occam programs are among the most famous applications of CSP. A Transputer is a fairly conventional microprocessor, with some hardware support for the Occam/CSP model of concurrency that would reside on the same chip with its required RAM memory. A number of Occam processes could share the same Transputer using a microcoded scheduler and employ local channels (memory) to exchange messages. Several Transputers would form a Transputer system by connecting together via four serial links, which correspond to Occam channels, used to send data in one direction and receive an acknowledgement back. These design features were dictated by the cost of integrated circuits and the interconnect between them. The same concept is found in MPPAs, with variation in the complexity of the processors and the type of the interconnect [4–6].

Ambric AM2045 [4], for instance, contains 336 32-bit processors, organized in 5×9 *brics*, and offers three levels of communication. Along with the intra-bric communication (level 1) including crossbars between the bric's four CPUs, there are also bric-to-bric channels (level 2) that map one-to-one to channels, as well as a global mesh interconnect (level 3) that can be shared by several channels. As in the Transputer, Ambric processors execute a single *object* (written in a subset of Java), and block whenever a channel send/receive cannot proceed. Ultimately, the intention is to achieve a high degree of parallelism at thread level and a scalable architecture.

From the architectural point of view, using networks on-chip (NoC) has become the answer to many of the scalability problems of shared memory [15,16]. Hybrid approaches, that efficiently support both shared-memory and message-passing programming paradigms, have recently been revived to fit systems on-chip [17,18]. The majority of NoC research focuses on mesh networks, due to their flexibility and regularity.

For instance, the Single-chip Cloud Computer (SCC [18]) is a recent experimental platform from Intel, comprising 24 tiles of two P54C cores, connected through a mesh network and capable of managing 16 GB of off-chip memory through four shared DDR ports. Core-to-core communication may occur either through the off-chip memory, or on-chip via message passing through the mesh network, that supports eight virtual channels. A similar approach is taken by Tilera's Tile64 processor [19], which is a 8×8 array of VLIW processors, connected through five independent mesh networks. Off-chip connections include four DDR ports, accessible to all cores. Both of these architectures are supported by C/C++ compilers and can run fully-fledged operating systems, such as Linux. Whereas the previous approaches avoid handling cache coherency issues and employ NoCs to directly exchange data between nodes, the newer TilePro64 architecture from Tilera as well as the work in [20] have dedicated mesh networks for supporting cache coherency.

In this paper, in contrast to the aforementioned work, we will adopt ring networks as on-chip interconnect, because of their predictability and simplicity required by our intended area of application, namely embedded real-time systems. In this sense, the communication infrastructure we adopt is perhaps most similar to the mesh network in [21], yet less complex and therefore using fewer resources. Furthermore, our architecture supports direct Java bytecode execution, offering a Java virtual machine view to the programmer.

With the advent of Java as a programming environment and building upon the success of the CSP formalism, a number of Java libraries supporting the CSP semantics have been developed. Among these, Communicating Sequential Processes for Java (JCSP, [11]) and Communicating Threads in Java (CTJ, [12]) appear to be the most mature. JCSP and CTJ co-evolved and have many similarities, providing a full range of CSP constructs, with JCSP focusing on general concurrent programming and CTJ offering support for real-time systems [22]. In both approaches channels are supported on shared memory or by explicitly extending classes for embedded to communication hardware. JCSP does provide a networking package (JCSP.net), intended for communication between different JVMs, but this seems to have a rather high communication overhead in terms of number of threads and delay [23]. Our intention is to learn from all these approaches, instead of competing with them, and gather the best features for our system, as we detail next.

3. CSP for a Java CMP

The Java chip multiprocessor [24,9] we focus our attention on is designed for embedded systems, contains a number of processors

sharing a global memory, and is intended for FPGA platforms. The processors are Java processors, named JOP, which are an implementation of the Java virtual machine in hardware [25]. The main design constraint of JOP is time-predictable execution of Java bytecode. Therefore, it is an easy target for worst-case execution time analysis [26,27]. We use a Java processor, as this is the only solution we are aware of, where WCET analysis for Java is possible.

JOP supports real-time multitasking, using a preemptive and strictly priority based scheduler. The real-time Java version that JOP supports is aiming towards the upcoming specification of safety–critical Java [28]. On a CMP system, threads are pinned to individual processor cores; threads do not migrate between cores. That means, that each core executes a preemptive, priority based scheduler.

To keep this CMP system time-predictable, the access to the shared main memory is controlled by a TDMA-based arbiter. The static schedule of the TDMA arbiter has been integrated into the low-level timing model of JOP in the WCET analysis tool WCA [26], thus WCET analysis is possible even for a JOP-based CMP system. Fig. 1 shows a JOP CMP system with TDMA based access to the shared memory. Each processor core also contains local memories: M$ is a method cache for instruction caching, S$ is a stack cache for stack allocated data, and SPM stands for a scratchpad memory. These three local memories contain only thread local data and need no cache coherence protocol. JOP may also be configured with an object cache [29], which supports cache coherent sharing of data. However, as we want to explore CSP based communication, we have disabled the object cache in our experiments. Nevertheless, the communication between threads is carried out through the shared memory. Naturally, for systems with more than two processors, the arbitration for shared memory accesses is increasing, and leads to a bottleneck in execution. In the evaluation we employ the scratchpad memory for local operations.

Our solution is to employ a dedicated communication structure between processors, along with fast local memories, and a CSP programming approach in order to relax the memory bandwidth pressure. A number of restrictions and limitations made us to adopt our own communication hardware and software solutions instead of using existing ones.

First, non-local CSP channels should map to dedicated hardware, as in Transputers, instead of shared memory as in JCSP and CTJ. These hardware links should be efficient, parallel rather than serial, and at the same time-shared between different CSP channels. The structure should be scalable and easily accommodate a larger number (tens) of processors, with minimal overhead due to limited resources available in embedded systems. Second, the software support should also be efficient, keeping the threads and context switches to a minimum. Furthermore, the processors uses Java, thus approaches that compile Java to native calls to specific kernels (i.e. [30]) are not applicable. Also, using channels should be transparent, regardless of where the communication processes are located (same processor, different processors on the same chip or different chips). Third, maintaining the real-time characteristics of the system is an important aspect, calling for predictable hardware and software behavior.

4. Hardware support for CSP

It is possible to implement CSP channels via shared memory, but that would undermine our primary goal of relaxing the memory bandwidth pressure in our CMP system. Avoiding shared memory entirely; Transputers and MPPAs employ mainly point-to-point links to exchange data between neighbor processors, complemented by a global network. Typically in such architectures point-to-point links map one-to-one to CSP channels, thus links are exclusively used by single processes. This limits the mapping of processes to processors and may lead to underutilized hardware resources, making such approaches desirable mainly for streaming applications, with regular flow of data. For embedded systems with limited resources and lower requirements in performance, a balanced solution is possible.

4.1. A basic TDMA network

In our case, we decided to adopt a ring network-on-chip (NoC) that can accommodate several virtual CSP channels on a single physical connection, in a Time Division Multiple Access (TDMA)

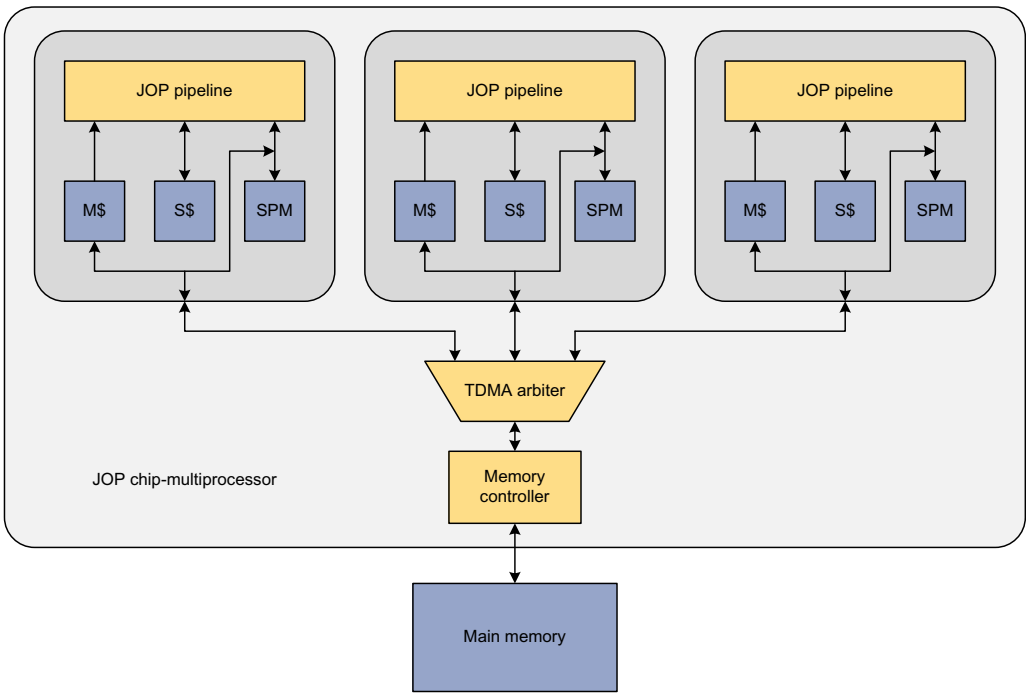


Fig. 1. CMP system based on JOP.

manner. This concept is similar to the proposed time-triggered NoC [31], the Æthereal mesh NoC [21], or the static scheduled real-time NoC [32]. Compared to the aforementioned approaches, we have further simplified the scheduling of packets. Each sender has pre-assigned a unique send slot, thus no scheduling tables are needed. While this pure TDMA approach is wasting link bandwidth, it results in a very simple router design. Therefore, the savings in chip area due to the simple router design, avoidance of buffers and flow control at the routers, may lead to a better bandwidth/cost tradeoff than trying to add best effort routing on top of a time-predictable NoC.

The network itself is composed of a ring of N registers for N nodes. An example of such a simple NoC for four JOP nodes is shown in Fig. 2. The registers shift every clock cycle.[1] Every Nth clock cycle, the information repeats – unless modified by the nodes. Each node has allocated a single slot (clock) where it can send data to any of the other nodes. Note that when a processor does not use its slot, no other processor can use that slot, i.e. there is no dynamic bandwidth allocation. Each slot (*phit* in our case) carries its own identifier, the destination identifier, a type (NIL, DATA, EOD, Ack), and a load of one word. Slot identifiers are constant and cannot be overwritten, while the rest of the contents vary. Note that this is also the *flit* unit for our network.

Packets can be composed of an unlimited number of flits, each flit being acknowledged on individual basis. Thus, there are never more than one flit en route from a source to a destination, and no re-ordering of flits or packets is needed. Messages, for our purpose may extend over several packets or be exactly one packet, containing one flit only.[2]

As an example of how this setup works, suppose node P needs to send M words to node R. P will be able to send DATA flit every Nth clock, once its slot comes around. When P realizes that itself is the destination from slot P, R starts receiving information (paying attention only to P slots) until a special EOD – end of data – word is received. EOD type flits may contain data (in case $M = 1$). For every received word, R alters the content of slot P to an ACK. Notice that R can write in slot P, but only acknowledgement flits (ACK). When the ACK reaches back to P (P listens to P slots when sending data), P is ready to send the next word in the packet. If P detects that the flit is not an ACK, it simply lets it spin. If P detects an ACK to its last flit, it fills frame P with a NIL flit. Whenever a node has nothing to send, it fills its own slot with a NIL flit. If a receiving R detects an ACK flit in the P slot, it simply lets it spin. In fact, in all other situations than acknowledging a new flit, R leaves it go through unmodified.

Note that each node can use only its own slot to send data (DATA, EOD). Destination nodes use the source node slot to send the ACK. As the mapping of slots to nodes is one-to-one, and each flit needs an ACK, there is no need for sequence numbers.

A block diagram of a simple router implementing this protocol is depicted in Fig. 3. Routers can handle the network communication independently, exchanging data with a JOP processor whenever needed, through a SimpCon interface [33]. If buffers are full/empty in the destination/source, the flits spin around unmodified until buffer space/data becomes available.

Multiple clock domains and support for dynamic voltage and frequency scaling may be introduced by using asynchronous FIFOs in the router and synchronizers for the registers read/written through SimpConIF, similar to the approach in [31]. The actual ring would thus function in one clock domain, while all the nodes in their own clock domain.

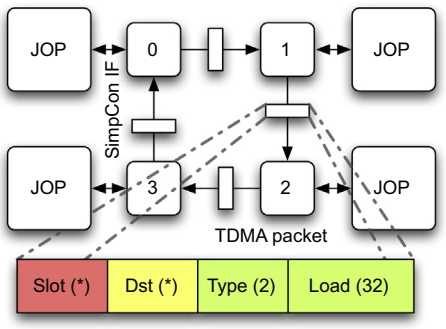


Fig. 2. A ring NoC with four simple routers.

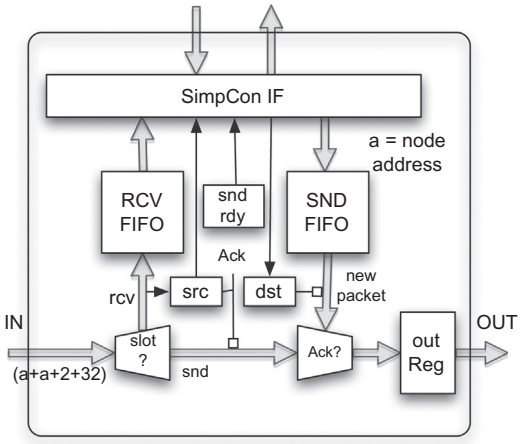


Fig. 3. A simplified block diagram of our NoC router.

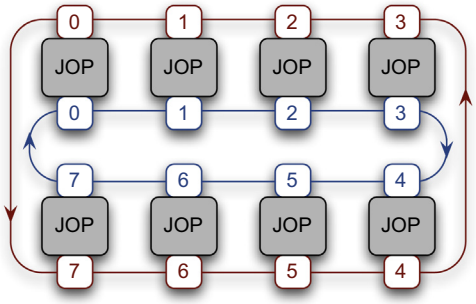


Fig. 4. A NoC of two overlapped rings. The numbers in the boxes are node addresses.

4.2. More complex networks

The routers presented above were designed for simplicity and speed, to allow fast message exchange between processors, while at the same time being highly predictable. Taking into account only performance and scalability, ring networks are fare worse than mesh networks [34]. For N nodes, the diameter and bisection width in a 2D mesh are $2(\sqrt{N}-1)$ and N, while in a uni-directional ring these are $N-1$ and 1. Nevertheless, in a mesh, even with a number of virtual channels, packet latency is sensitive to cross traffic due to

[1] Naturally always shifting the data in the ring consumes power. Some power-aware techniques we experimented with are: (a) using gray code node addresses, (b) avoiding shifting the load for non-data flits, (c) stalling the ring when no progress has been made for N clocks. The results and discussions related to these techniques are out of the scope of this paper.

[2] To summarize, the ring network has a phit size of $32 + a + a + 2$ bits (a being the address field size), a flit size of one phit, while packets and messages may be arbitrarily long.

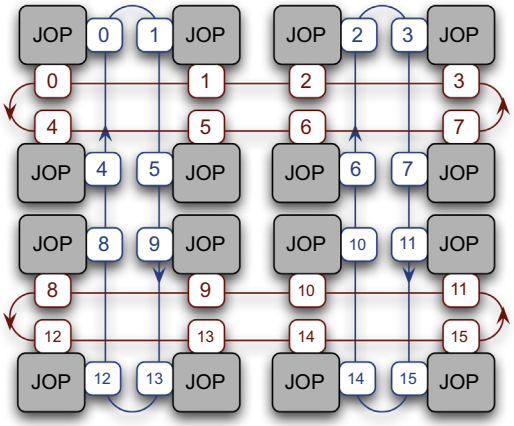


Fig. 5. A NoC of multiple rings connecting a 2D array of processors.

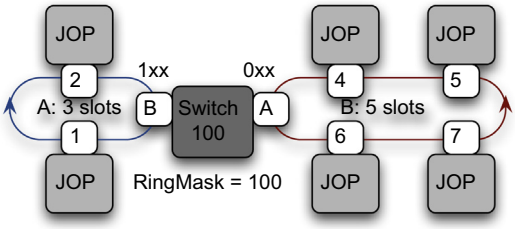


Fig. 6. A NoC of two rings connected through a switch.

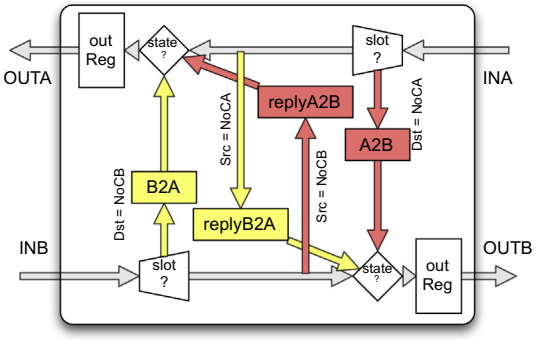


Fig. 7. Block diagram of the NoC switch between rings.

route contention [35] unless special measures are taken to control or even avoid contention [21]. The TDMA ring we adopt herein is less sensitive to such traffic, and therefore more predictable, thus suitable to the type of systems we target. In the worst case, when all nodes decide to send packets to the same destination, one node must wait for at most $N-2$ packets to arrive at the destination, before sending its packet. Reducing the packet size decreases this worst-case latency.

Nevertheless, if network performance becomes a problem for a single ring, more complex configurations can be formed, either using several rings, if predictability remains an issue, or combining rings with a mesh or other structures. We present solutions that remain highly predictable in the following.

One possibility is for every JOP node to use two or more routers, in order to connect to several rings at the same time. For example, a multiple rings architecture similar to the CELL processor [36] would be possible (see Fig. 4). Another option is a 2D ring-mesh (similar to a Torus), as shown in Fig. 5. Packet routing and switching between the different rings must be, in this case, handled in software. This results in more complex software in addition to the two routers per node cost in hardware.

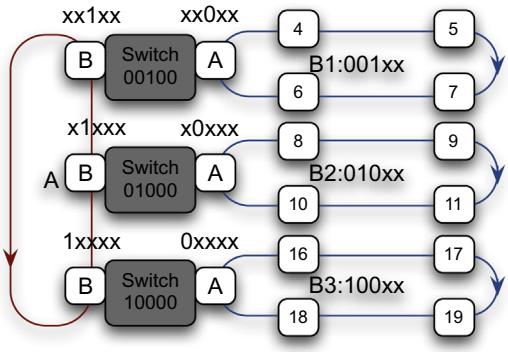


Fig. 8. A NoC of three rings connected through another ring and three switches. The processors are omitted for clarity.

To overcome the increase in complexity introduced by the multiple routers per node, we propose another network architecture, based on multiple rings interconnected through hardware switches. Furthermore, the rings may be of different size, and the node assignment such that the communication delay is minimized. This allows building hierarchical systems where local networks provide high bandwidth and short latency and the next level connects those local networks. An example of such a network is depicted in Fig. 6, which contains two rings, A and B, connected through a hardware switch. The nodes are still using a single router, and are unaware of the fact that they are possibly located on different rings. The switch introduces its own slot in both rings; slots that play the roles of gateways from ring A to B and B to A respectively. The slot identifiers and node addresses are initialized in such a way that nodes on ring A will use the slot of the switch to receive data from (and acknowledgements to) B. When sending data from an A node, the switch picks up on the destination address being from B, and forwards this data in its own slot from the B ring. Additional simultaneous requests will continue to cycle the sending ring until the current send is acknowledged and the switch is able to handle the next pending request.

Unlike the local nodes' slots, the switch slots source address may change at runtime, according to which remote node is using the switch at that time. For example, in Fig. 6, if node 1 sends a package to node 5, the slot corresponding to the switch in ring B will have its identifier set to 1. Likewise, if node 2 now sends a package to node 7, the same slot will change its identifier to 2. Nevertheless, the switch will always be identified through its 0 leftmost bit, which does not appear in the addresses of the local nodes. All of the local slots in ring B will always keep their identifiers as 4, 5, 6, and 7. The switch is built such that it allows each of the nodes from A, in a round robin fashion, extend their slots through B as well, allowing nodes from B to receive and acknowledge flits from A. The behavior is similar when sending flits from B to A. The switch implementation was kept simple, comprising two frame registers and a four states FSM for either of the A-to-B and B-to-A loops, as depicted in the block diagram from Fig. 7.

The main advantage of a multi-ring network architecture is the ability of reducing the communication time between certain nodes at the expense of the communication time between other nodes. In particular, nodes that exchange large amounts of data should be placed on the same (preferably small) ring, allowing for occasional exchanges with remote nodes via switches. Another example of a multi-ring network is depicted in Fig. 8, where three different rings of four nodes are connected via a fourth ring of switches only.

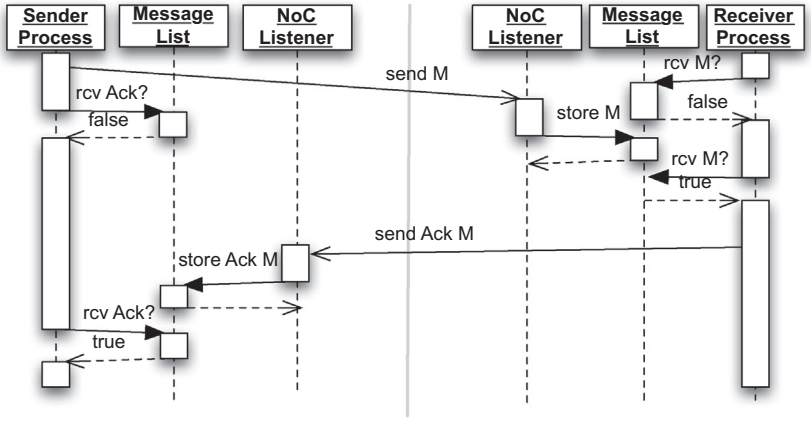


Fig. 9. Sequence diagram for rendezvous with an explicit Ack message.

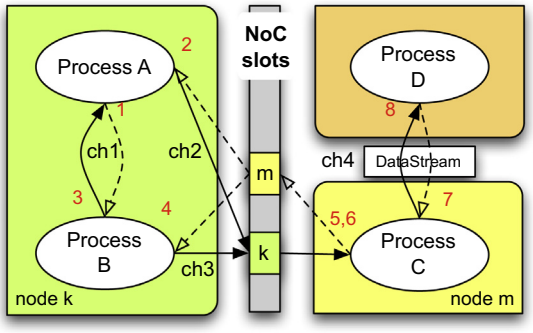


Fig. 10. An example of four CSP processes running on three processors, communicating through four channels of various types. The dashed lines represent the Ack message path.

Table 1
Configurations used in the experiments. * Marks the maximum for that FPGA.

ID	Board	FPGA	JOP cores	Clock (MHz)	Off-chip RAM (KB)	SPM (KB)
3ne2	Digilent Nexys2	XC3S1200e	3*	50	4096	1
3cyc	Jopdesign Cycore	EP1C12	3*	40	1024	1
8de2	Altera DE2-70	EP2C70	8	50	2048	1
4de2	Altera DE2-70	EP2C70	4	50	2048	1

Table 2
Resource usage for two CMP systems with the proposed NoC on Altera DE2-70.

Modules	4de2		8de2	
	Logic	Memory	Logic	Memory
All processor cores	14427 LC	38 KB	39000 LC	76 KB
All I/O modules	1236 LC	0 KB	2268 LC	0 KB
Memory arbiter	705 LC	0 KB	1381 LC	0 KB
TDMA NoC	2008 LC	0 KB	4168 LC	0 KB
Total	22975 LC	38 KB	46560 LC	76 KB

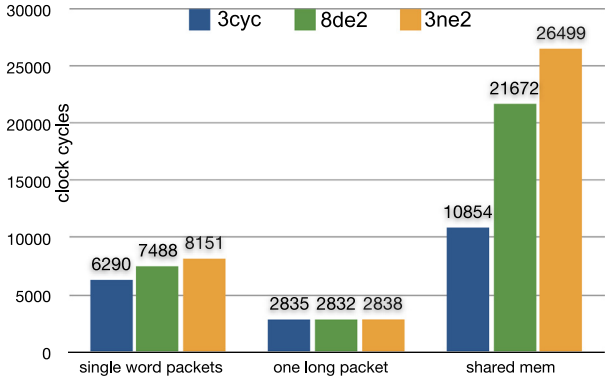


Fig. 11. Transfer time in clock cycles for 100 words via NoC and shared memory.

Multiple clock domains would be more naturally introduced at the ring-switch level, where different rings would operate in their own clock domains, and the ring-switch takes care of the clock domain crossing.

Computing the communication latency in a multi-ring network is more complex than in a single ring, from several reasons, as follows. Packages circulate with different rates in different rings, depending on the number of nodes in each ring. For example, in Fig. 6, the switch can send data in ring A every 3rd clock cycle, while in ring B only every 5th cycle. Another issue results from the switch being shared between all the nodes in a ring, and in the worst case a node may have to wait for all the other nodes to carry out their round through the switch. In this context, taking again Fig. 6 as example, if one A node (and no other) needs to send a package to a B node, the round trip (required for the acknowledgement) will take $2 * (N_A + N_B)$ clock cycles in the worst case, where N_A and N_B are the number of nodes (slots) in each ring.[3] This accounts for the switch delay (one clock cycle in each direction) and the possibility that the package just missed its slot when switching rings (due to rate difference). For a single ring configuration of the same nodes, minus the switches, the delay would amount to $N_A + N_B - 2$. However this will be between any two nodes, while for the dual-ring network, the intra-network delays are only N_A and N_B respectively.[4]

This speaks for the need of a proper partitioning of the functionality between nodes, such that tasks exchanging a lot of data often should be mapped to the nodes within the same ring.

[3] This can be combined with the worst case scenario that each node in A may wish to send a package at the same time, leading to a worst case delay of $N_A * 2 * (N_A + N_B)$.

[4] For rings operating in different clock domains this analysis would be even more complex, since it needs to account for additional synchronization overhead.

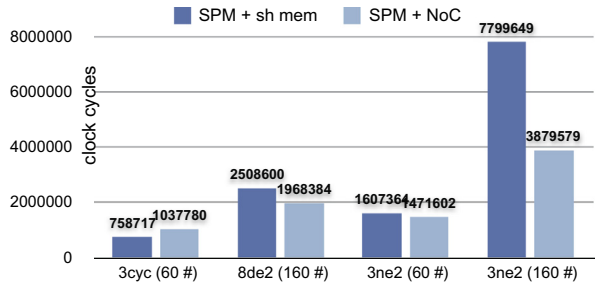


Fig. 12. Computing primes using Eratosthenes' sieve for various CMP configurations.

4.3. Core local private memory

To avoid traffic on the shared memory it is of utmost importance to use core local, on-chip memory. The common solution is caching. However, caches with their cache coherence protocol are also a scalability issue. A scalable solution is to use explicit local memories, called scratch-pad memories (SPMs). As the CSP based communication uses channels instead of cache coherent memory, the SPM fit very well to the hardware based CSP implementation. To avoid invalid sharing of objects allocated in core local memories the SPM is mapped to *private memory*, which is an instance of a RTSJ style scoped memory [37]. The private memory is used for buffers to receive and send channel messages and store the data for processing. In the current implementation the processor transfers the data between the NoC and the private memory. In a future implementation one might use an additional read/write port, which is available in modern FPGAs, to allow a DMA based transfer of the NoC data directly to the private memory.

4.4. Network software interface

A NoC router is connected to a JOP node through a SimpCon interface [33], thus its registers are memory mapped. Communication with the router is achieved through four addresses that can be read or written. Read addresses have the following meaning:

00 *StatusReg*: Uses the lower 16 bits. The lower byte is the node address (set at synthesis). The upper byte contains flags describing the busy state of the router, EoD seen flag, and the send/receive FIFOs full/empty status. These depend only on the FIFO size, not the packet size. Default FIFOs are of size two.
01 *not in use*[5]
10 *RcvSourceReg*: The source address for this packet.
11 *RcvDataReg*: Next word from the packet. Blocks if the receive FIFO is empty. To check the status use the StatusReg, for nonblocking operations.

Write addresses have the following meaning:

00 *RcvReset*: Resets EoD and the receive FIFO, allowing new incoming messages to be received.
01 *SndCntReg*: Use to specify the number of words in the packet. Clears the send FIFO. Sending starts automatically after this.
10 *SndDestReg*: Use to specify destination node address.
11 *SndDataReg*: Use to send the next word in the packet. Blocks if send FIFO is full. For non-blocking operations check the StatusReg.

Typically a reception can start at any time from any non-masked source. Once the reception started, that source is followed until the EoD is received. The software has to keep reading data from the receive buffer until empty and the end of packet flag is set. No new receptions may start unless the end of packet flag is reset explicitly through software. To send a packet, the destination must be provided first, followed by the number of words in the packet and then the content of the packet. The software should monitor the send and receive buffer flags, to avoid attempts to write in a full send buffer or read from an empty receive buffer, both of which block until the operation can be carried out.

5. Software architecture

In hardware, sending a message between two nodes is acknowledged on word basis. This means that the sender and the receiver nodes do have a rendezvous in the CSP sense. Transputers and MPPAs in general require a rather restrictive mapping of processes and channels to processors and links. Among other restrictions, each link (often four for each node) is dedicated to one channel only. Using the same restriction for our architecture, a basic hardware network operation (one send or one receive) is enough to achieve the rendezvous behavior required by CSP. In fact, the sender and the receiver will be slightly out of phase (the receiver continues after an EoD flit, while the sender continues after the Ack to its EoD flit), by exactly K clock cycles, if there are K hops in between the receiver and sender (network distance).

5.1. Sharing processors and channels

Our intention is to allow both several processes to share the same processor and several channels to share the same slot. This allows more flexibility in mapping applications to our architecture. The real complications appear from channels in between different processes sharing the same slot. One idea would be to block any other outgoing communications as soon as a task needs to send (receive), until the other end is ready to receive (send), so as to achieve a true CSP rendezvous. It is essential to realize that the sender and receiver of the same channel must be selected for execution on the communicating nodes. Note that it is very possible that the sender (receiver) may block other processes from using the NoC for an indeterminate amount of time. This may lead to deadlocks or inefficient execution at best.

Another way, which we adopt herein, is to receive any incoming messages in a system task, but have the individual receiver acknowledge them explicitly through a message back to the sender (as depicted in Fig. 9). Senders in their turn, must await and receive this acknowledge. The physical medium is in this case not occupied waiting for a receiver to start receiving, allowing several virtual channels to communicate at the same time, in any order. The drawback is the need for the additional acknowledgement at message level. The rendezvous behavior is kept, while also allowing for better concurrency.

5.2. Implementation details

We have written a CSP Java library (11 classes currently) that implements channel communication as described above. It allows for different types of channels (local, NoC, and stream), and provides the same interface to all. Local channels are supposed to be used by processes running on the same processor. NoC channels

[5] In fact *deprecated*, since in early versions it was used to signal which slots contain data that can be received. Along with a slot mask, this was initially intended for implementing Occam-like ALT and PRI ALT, but this functionally is better handled in software when multiple threads may share the same physical resources.

are supported through the NoC hardware, as a means of communication between processors on the same chip. Stream channels use the standard Java *DataStream* classes to implement CSP channels and are intended for off chip communication, such as TCP/IP or RS232. Regardless of the type, all the received messages are handled by the same message queue, which is a shared resource between all the processes and the system tasks listening to communication media (i.e. the NoC Listener in Fig. 9).

Since channels are now bi-directional (the ACK message needs to get back to the sender[6]), we assign a unique identifier to each channel end. Messages going through the channels are carrying the destination end identifier with them, so that they can be found in the common message queue. Setting up a channel requires creating two ends, one used to send (receive) the actual message and the other used to receive (send) the acknowledgement. Ends must be of appropriate type, depending on where the processes are located relative to each other.

An illustrative example of four processes executing on three processors, communicating through four channels of various types is shown in Fig. 10. Channel 1 is a local channel, channels 2 and 3 are NoC mapped channels, while channel 4 is a stream channel. The red numbers represent the channel end identifiers.

Using Java threads as CSP processes is an obvious choice that we also adopt at this stage in our approach. In particular, on each processor in a CSP system, there is one Java thread per CSP process along with one listener thread for each shared communication media (one NoC listener, plus one listener per stream). Local channels do not require listeners. Compare this to JCSP networking, which requires more than six processes/threads [23] per channel.

5.3. Further development

The CSP library developed for JOP CMP is rather limited at this point, focusing on providing basic functionality. The intention is to make this more complete, to offer more support to the programmer. Typed channels are easy to introduce, by employing Java serialization or a JDO-like persistence [38], over the existing channels. CSP constructs similar to the Occam ALT, PRI ALT, PAR, PRI PAR, although possible through regular Java constructs, will be added for better support.

Channel creation at this point requires exact knowledge of the location and type of channels. Additionally, processes are explicitly bound to processors by the programmer, just as in Transputer and MPPA programming approaches. Our intention is to build a source pre-processor that can merge processes, optimize the process to processor mapping and channel assignment, generating code that uses the CSP library.

6. Evaluation

The TDMA routers, switches, and NoC have been implemented in VHDL and tested separately before integrating them with the CMP. The critical path delay was kept under the delay in the JOP critical path, and synthesis results for full systems show no performance penalty in terms of clock speed.

Various JOP-based CMP systems were synthesized and tested on three different FPGA platforms, 3-core versions on the Jopdesign Cycore (Cyclone EP1C12, fast SRAM) and the Digilent Nexys2 boards, as well as up to 8-core versions on the Altera DE2-70 board. A summary of configurations employed for the experiments described in this paper is given in Table 1.

[6] In fact the ACK message does not have to get back to the sender through the same medium, but it makes sense to do so.

6.1. Resource consumption

Table 2 shows the resource consumption of the component groups for a 4-core (`4de2`) and 8-core (`8de2`) versions on the Altera DE2-70 board. Note that the TDMA-based NoC consumes less than a single processor core, even for the 8-core system. Relative to the whole design the resource consumption is between 8.7% and 8.9%, for the 4-core and the 8-core systems respectively. Similar relative device utilization figures were observed on the two other FPGA platforms.

The resource usage is smaller than in typical mesh networks. For instance, in Hermes [39], a wormhole/packet switching NoC template, the switch uses 631 4LUTs and 200 FFs and the send/receive interface takes 193 LUTs and 233 FFs (4LUTs on Virtex2, buffer size of eight) for 8-bit flits. Compare these figures to the data for our router taking 450 LUTs and 176 FFs (4LUTs on Spartan3e) for five times larger flits of 40-bits in our case.[7]

6.2. Raw performance

In order to compare the performance of the CSP exchange via the NoC versus using the shared heap, we measured the time required to transfer data from one processor to another, using shared memory, short NoC packets (1 word/packet) and long NoC packets (100 words/packet) for a number of CMP configurations and FPGA platforms. The shared memory communication is performed via external, low latency SRAM and therefore not cached. The overhead of the CMP SRAM access comes from the time-predictable arbitration in a TDMA fashion. The same would be true if a shared cache needs to be accessed time-predictable.

Fig. 11 shows the transfer time, in clock cycles, of 100 32-bit words between processors for the different configurations. On all of the boards used, the NoC-based communication is considerably faster than the shared memory approach. For instance, on the `3cyc` configuration, communicating 100 words (1 word/packet) via NoC is 1.7 times faster than using the on-board fast SRAM memory. For the `8de2` configuration, due to the increase in number of cores that need to share the main memory, the speed-up is larger at 2.9. For the `3ne2` configuration, due to its slow pseudo SRAM-based main memory, the speed-up of the CSP communication is 3.3 for the 3-core setup. These figures increase to 3.8, 7.7 and 9.3 respectively, when long packets (100 words/packet) are used, since the header overhead is reduced.

Additionally, we also observed that the communication delay over the NoC scales linearly with the amount of data. Nevertheless, the size of the NoC seems to have almost no effect on the communication delay. We observed a similar behavior when looking at switched dual ring NoCs in 3-core CMP, where both intra-ring and inter-ring communications are similar in performance to single ring NoC.

In both cases, this effect comes from to the minimal increase (a few clock cycles) in the network latency, which remains unnoticeable due to buffering and relatively slow processors. For larger systems, the network latency may however increase to the point where it may have a noticeable effect.

6.3. Application performance

To gather further insight into the performance impact of the on-chip communication we have created a micro benchmark and adapted an embedded benchmark.

[7] Although a fair comparison is hard to make, we consider these setups to be roughly equivalent, since a Hermes router has five 8-bit wide ports, while our network has one 40-bit wide port for the ring plus another 32-bit wide port connecting to the processor.

The micro benchmark calculates prime numbers with a distributed version of Eratosthenes' sieve algorithm. One processor generates the input (increasing numbers), while the rest receive numbers and forward them to the next processor only when they are not multiples of a prime. Ideally each processor would wait for numbers, and as soon as one receives the first number (which is unfiltered by the previous processors, meaning it is prime) it starts acting as a filter. Naturally, the number of processors is limited, and therefore in our implementation each of the N processors holds a list of every *N*th prime, acting as an overlap of single filters (one for each time a number comes around). Execution times (clock cycles) for computing the first 60 and first 160 primes for few configurations are depicted in Fig. 12. For these experiments we used core local memory (`SPM`) for computations and shared memory (`sh mem`) or NoC (`NoC`) to forward candidate numbers between processors. From the figures we observe that as the shared memory latency and the amount of data increase, so does the speed-up obtained from using the NoC versus shared memory.

As a more realistic benchmark we have looked at the AES benchmark from the embedded Java benchmark suit JemBench [40]. AES is a pipelined application that generates data, encrypts the data, decrypts it again, and checks the output for correctness. The original benchmark uses a non-blocking queue of buffers for the exchange of data. In order to explore the full benefit from the fast on-chip communication we adapted the benchmark to use the core local memories. Additionally, to make a fair comparison, we also adapted the original shared memory version to use core local memories for the local computations. We carried out experiments with AES on the 8-core `8de2` configuration, where each stage runs on its own processor, and on a 3-core `3ne2` configuration, where data generation and the check for correctness are carried out together on one processor, while encryption and decryption are separately assigned to the remaining processors.

The measurements on the `8de2` do not show any significant speed-up, while the `3ne2` show a decrease in computational speed for the NoC/CSP version. This is however not unexpected, as AES is a benchmark that would benefit from executing computation and communication in parallel, which is not the model of computation of CSP. For this kind of pipelined computation CSP may in fact introduce unwanted synchronization, leading to performance loss. However, note that CSP is a cleaner abstraction for concurrent programming and the performance is on par with the original more ad hoc parallel implementation. From that experiment we conclude that we shall explore additional models of computation, such as synchronous data-flow (SDF), which need additional software implementation. The hardware NoC implementation can be used for SDF.

7. Conclusion

We have presented a hardware support for a CSP implementation on a embedded Java CMP. The on-chip communication channels are introduced in order to reduce the bandwidth pressure on the shared memory. Our solution is specific for systems with limited resources and real-time requirements. Ring-based networks-on-chip along with scratch-pad memories are used to implement fast and time predictable communication between processors. Both single and multi-ring networks are feasible using the routers and switches introduced in this paper.

The concept was tested on CMP with different configurations, from three to eight JOP processors, implemented on three different prototyping boards, with both Altera (EP1C12, EP2C70) and Xilinx (XC3S1200e) FPGAs. Synthesis results show that the device area overhead for introducing a NoC is in the 9% range, without any performance penalty. Measurements of the raw speed-up obtained using the NoC versus shared memory to transfer data between processors are between 1.7 and 9.3 depending on the packet size and the shared memory latency. When it comes to application speed-up, there is a large variation in behavior for different applications, depending on the type of processing involved. In the worst case, the tight synchronization required by the CSP programming model may hurt applications that are better suited for SDF-like processing (e.g., pipelined computations). To conclude, only certain applications can benefit from using the CSP approach from the performance point of view.

In future work we intend to explore many-core systems in a large FPGA with use the multi-ring architectures. We expect that the main challenge is the partitioning of the application on this system.

The whole design is provided under the GNU GPL open-source license to support future research on the topic of hardware support for CSP style on-chip communication.

8. Source access

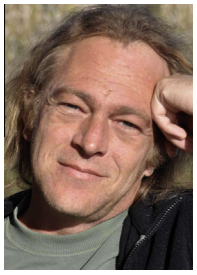
The source files for the hardware and software used in this paper can be found in the JOP archive. The download procedure is described at http://www.jopwiki.com/Download. Follow the `make` instructions to obtain the automatically generated `vhdl` modules required. Relative to the archive root, the Altera top level VHDL modules for the Cycore board and the DE2-70 are located in `vhdl/paper/csp`, while the Xilinx ISE 12.2 project for Digilent Nexys2 board is located in `vhdl/paper/nexys2_csp`. Synthesize and download the hardware using the appropriate tools for your board. The software support for CSP is located in the `java/target/src/paper/csp` directory.

References

[1] C. Whitby-Strevens, The transputer, SIGARCH Comput. Archit. News 13 (3) (1985) 292–300, http://dx.doi.org/10.1145/327070.327269.
[2] M. Homewood, D. May, D. Shepherd, R. Shepherd, The IMS t800 transputer, IEEE Micro 7 (5) (1987) 10–26, http://dx.doi.org/10.1109/MM.1987.305012.
[3] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, 4th ed., Morgan Kaufman Publishers, 2006.
[4] M. Butts, Synchronization through communication in a massively parallel processor array, IEEE Micro 27 (5) (2007) 32–40, http://dx.doi.org/10.1109/MM.2007.92.
[5] Intellasys, SEAForth 40C18 DataSheet, 9th ed. <http://www.intellasys.net/>.
[6] H. Svensson, Reconfigurable Architectures for Embedded Systems, Lund University, Lund, 2008 (diss. Lund: Lunds universitet, 2008).
[7] G. Kahn, The semantics of simple language for parallel programming, in: IFIP Congress, 1974, pp. 471–475.
[8] C.A.R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677, http://dx.doi.org/10.1145/359576.359585.
[9] C. Pitter, M. Schoeberl, A real-time Java chip-multiprocessor, ACM Trans. Embed. Comput. Syst. 10 (1) (2010) 9:1–9:34, http://dx.doi.org/10.1145/1814539.1814548. <http://www.jopdesign.com/doc/jopcmp_tecs.pdf>.
[10] P.H. Welch, J.R. Aldous, J. Foster, CSP networking for java (JCSP.net), in: P.M.A. Sloot, C.J.K. Tan, J. Dongarra, A.G. Hoekstra (Eds.), International Conference on Computational Science, Lecture Notes in Computer Science, vol. 2330, Springer, 2002, pp. 695–708. <http://link.springer.de/link/service/series/0558/bibs/2330/23300695.htm>.
[11] P.H. Welch, N. Brown, J. Moores, K. Chalmers, B.H.C. Sputh, Integrating and extending JCSP, in: A.A. McEwan, S.A. Schneider, W. Ifill, P.H. Welch (Eds.), The 30th Communicating Process Architectures Conference, CPA 2007, Organised Under the Auspices of WoTUG and the University of Surrey, Guildford, Surrey, UK, 8–11 July 2007, vol. 65 of Concurrent Systems Engineering Series, IOS Press, 2007, pp. 349–370. <http://www.booksonline.iospress.nl/Content/View.aspx?piid=5962>.
[12] G.H. Hilderink, J.F. Broenink, W.A. Vervoort, A.W.P. Bakkers, Communicating Java threads, in: 20th World Occam and Transputer User Group Technical Meeting, IOS Press, Amsterdam, Enschede The Netherlands, 1997, pp. 48–76. <http://www.ce.utwente.nl/javapp/cjt/CJT-paper.PDF>.
[13] F. Gruian, M. Schoeberl, NoC-based CSP support for a Java chip multiprocessor, in: Proceedings of the 28th Norchip Conference, IEEE Computer Society, Tampere, Finland, 2010. <http://www.jopdesign.com/doc/csp_on_jop.pdf>.
[14] D. May, R. Shepherd, Occam and the transputer. In: Proc. of the IFIP WG 10.3 Workshop on Concurrent Languages in Distributed Systems: Hardware

Supported Implementation, 1985, Elsevier North-Holland, Inc. New York, NY, USA, pp. 19–33.
[15] X. Wang, G. Gan, J. Manzano, D. Fan, S. Guo, A quantitative study of the on-chip network and memory hierarchy design for many-core processor, in: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2008, pp. 689–696. http://dx.doi.org/10.1109/ICPADS.2008.18. <http://portal.acm.org/citation.cfm?id=1491261.1491566>.
[16] M. Forsell, Performance comparison of some shared memory organizations for 2d mesh-like NoCs, Microprocess. Microsyst. 35 (2011) 274–284, http://dx.doi.org/10.1016/j.micpro.2010.07.003.
[17] M.R. Casu, M.R. Roch, S.V. Tota, M. Zamboni, A NoC-based hybrid message-passing/shared-memory approach to CMP design, Microprocess. Microsyst. 35 (2011) 261–273, http://dx.doi.org/10.1016/j.micpro.2010.09.006.
[18] Intel Labs, Intel Corporation, SCC External Architecture Specification (EAS), Revision 0.94 Edition, May 2010.
[19] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, Tile64 – processor: a 64-core soc with mesh interconnect, in: I. International (Ed.), Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers, 2008, pp. 88–89.
[20] P. Sam, Investigating the Scalability of tiled Chip Multiprocessors Using Multiple Networks, Ph.D. Thesis, University of Manchester, 2009.
[21] K. Goossens, J. Dielissen, A. Radulescu, A ethereal network on chip: concepts, architectures, and implementations, Design Test Comput. IEEE 22 (5) (2005) 414–421, http://dx.doi.org/10.1109/MDT.2005.99.
[22] P.H. Welch, A.W.P. Bakkers, N.C. Schaller, Using java for parallel computing – JCSP versus CTJ, in: Communicating Process Architectures 2000, 2000, pp. 205–226.
[23] K. Chalmers, J.M. Kerridge, I. Romdhani, A Critique of JCSP Networking, in: F.R.M. Barnes, J.F. Broenink, A.A. McEwan, A. Sampson, G.S. Stiles, P.H. Welch (Eds.), Communicating Process Architectures 2008, 2008.
[24] C. Pitter, Time-Predictable Java Chip-Multiprocessor, Ph.D. Thesis, Vienna University of Technology, Austria, 2009. <http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1659>.
[25] M. Schoeberl, A Java processor architecture for embedded real-time systems, J. Syst. Archit. 54 (1–2) (2008) 265–286, http://dx.doi.org/10.1016/j.sysarc.2007.06.001. <http://www.jopdesign.com/doc/rtarch.pdf>.
[26] M. Schoeberl, W. Puffitsch, R.U. Pedersen, B. Huber, Worst-case execution time analysis for a Java processor, Software: Practice Exp. 40/6 (2010) 507–542, http://dx.doi.org/10.1002/spe.968. <http://www.jopdesign.com/doc/wcetana.pdf>.
[27] T. Harmon, Interactive Worst-Case Execution Time Analysis of Hard Real-Time Systems, Ph.D. Thesis, University of California, Irvine, 2009. <http://vocaro.com/trevor/files/Dissertation.pdf>.
[28] D. Locke, B.S. Andersen, B. Brosgol, M. Fulton, T. Henties, J.J. Hunt, J.O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, A. Wellings, Safety–critical Java technology specification, Public Draft (2011). <http://www.jcp.org/en/jsr/detail?id=302>.
[29] M. Schoeberl, A time-predictable object cache, in: Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2011), IEEE Computer Society, Newport Beach, CA, USA, 2011, pp. 99–105. <http://www.jopdesign.com/doc/ocache.pdf>.
[30] J. Moores, Native JCSP: the CSP-for-Java library with a low-overhead CPS Kernel, in: P.H. Welch, A.W.P. Bakkers (Eds.), Communicating Process Architectures 2000, Concurrent Systems Engineering, WoTUG, IOS Press, Amsterdam, 2000, pp. 263–273. <http://www.cs.kent.ac.uk/pubs/2000/1146>.
[31] M. Schoeberl, A time-triggered network-on-chip, in: International Conference on Field-Programmable Logic and its Applications (FPL 2007), IEEE, Amsterdam, Netherlands, 2007, pp. 377–382. http://dx.doi.org/10.1109/FPL.2007.4380675. <http://www.jopdesign.com/doc/ttnoc_fpl2007.pdf>.
[32] M. Schoeberl, F. Brandner, J. Sparsø, E. Kasapaki, A statically scheduled time-division-multiplexed network-on-chip for real-time systems, in: Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS), IEEE, Lyngby, Denmark, 2012. <http://www.jopdesign.com/doc/s4noc.pdf>.
[33] M. Schoeberl, SimpCon – a simple and efficient SoC interconnect, in: Proceedings of the 15th Austrian Workshop on Microelectronics, Austrochip 2007, Graz, Austria 2007. <http://www.jopdesign.com/doc/simpcon_austrochip2007.pdf>.
[34] L. Benini, G. Micheli, Networks on chips: technology and tools, The Morgan Kaufmann Series in Systems on Silicon, Elsevier, Morgan Kaufmann Publishers, 2006. <http://books.google.com/books?id=IHHTmSBcoGIC>.
[35] L. Tedesco, A. Mello, D. Garibotti, N. Calazans, F. Moraes, Traffic generation and performance evaluation for mesh-based NOCs, in: 18th Symposium on Integrated Circuits and Systems Design, 2005, pp. 184–189. http://dx.doi.org/10.1109/SBCCI.2005.4286854.
[36] M. Kistler, M. Perrone, F. Petrini, Cell multiprocessor communication network: built for speed, Micro IEEE 26 (2006) 10–25. <http://ieeexplore.ieee.org/iel5/40/34602/01650177.pdf>.
[37] A. Wellings, M. Schoeberl, Thread-local scope caching for real-time Java, in: Proceedings of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2009), IEEE Computer Society, Tokyo, Japan, 2009, pp. 275–282. http://dx.doi.org/10.1109/ISORC.2009.13. <http://www.jopdesign.com/doc/local_scopes.pdf>.
[38] Oracle, Java Data Objects. <http://java.sun.com/jdo/>.
[39] F.G. Moraes, N.L.V. Calazans, A.V. de Mello, L.H. Moller, L.C. Ost, HERMES: An Infrastructure for Low Area Overhead Packet-Switching Networks on Chip, Technical Report 034, PUCRS – Brazil (October 2003).
[40] M. Schoeberl, T.B. Preusser, S. Uhrig, The embedded Java benchmark suite JemBench, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010), ACM, New York, NY, USA, 2010, pp. 120–127. http://dx.doi.org/10.1145/1850771.1850789. <http://www.jopdesign.com/doc/jembench.pdf>.

Flavius Gruian is Assistant Professor at the Department of Computer Science of Lund University, Sweden, where he also received his PhD degree from. His research interest is in resource-limited embedded systems, Java processors, and energy-efficient scheduling. Flavius Gruian has published more than 15 refereed conference and journal papers.



Martin Schoeberl is Associate Professor at the Department of Informatics and Mathematical Modeling of the Technical University of Denmark. Before joining DTU he was Assistant Professor at the Institute of Computer Engineering of the Vienna University of Technology. His research interest is in timepredictable computer architecture and real-time Java. He is member of the expert group for the Safety–Critical Java Specification. Martin Schoeberl has published more than 70 refereed conference and journal papers.